Conference: ASPLOS 2009
Title of the paper: An Evaluation of the TRIPS Computer System
Authors of the submitted paper:

Mark Gebhart
Bertrand A. Maher
Katherine E. Coons
Jeff Diamond
Paul Gratz
Mario Marino
Nitya Ranganathan
Behnam Robatmili
Aaron Smith
James Burrill
Stephen W. Keckler
Doug Burger
Kathryn S. McKinley

An Evaluation of the TRIPS Computer System

Abstract

The TRIPS processor is designed to demonstrate a wide-issue large window microarchitecture while tolerating emerging technology scaling challenges such as increasing wire delays and power consumption. TRIPS employs a new instruction set architecture (ISA) called Explicit Data Graph Execution (EDGE) which renegotiates the boundary between hardware and software. EDGE ISAs use a block-atomic execution model in which blocks consist of dataflow instructions. This model preserves sequential memory semantics, enabling the system to expose high levels of instruction-level concurrency without a parallel programming model. Each TRIPS processor may execute up to 16 instructions per cycle from a window of 1024 instructions, using a distributed microarchitecture with small tiles that communicate via control and data networks. While many aspects of TRIPS have appeared elsewhere, this paper performs a detailed ISA and performance analysis to explore how well the hardware and software exploit the EDGE ISA. Compared to conventional ISAs, the block-atomic model increases concurrency at a cost of more instructions executed, and replaces register and memory accesses with more efficient direct instruction-to-instruction communication. We compare performance, using cycles counts, to commercial processors. On simple benchmarks, TRIPS outperforms the Core 2 by 40% and a factor of 3 on compiled code and hand-optimized code, respectively. On SPEC CPU2000, an Intel Core 2 outperforms TRIPS compiled code in most cases, although TRIPS roughly matches the performance of a Pentium 4. The lessons learned from the prototype point to several ISA, microarchitecture, and compiler adjustments that address the weaknesses of the current system.

1 Introduction

Growing on-chip wire delays, coupled with complexity and power limitations, have placed severe constraints on the issue-width scaling of conventional superscalar architectures. Because of these trends, major microprocessor vendors have abandoned architectures for single-thread performance and turned to the promise of multiple cores per chip. While many applications can exploit multicore systems, this approach places substantial burdens on programmers to parallelize their codes. Despite these trends, Amdahl's law dictates that single-thread performance will remain key to the future success of computer systems [7].

In response to semiconductor scaling trends, we designed a new architecture and microarchitecture intended to extend single-thread performance scaling beyond the capabilities of superscalar architectures. The TRIPS system represents the first instantiated prototype of theses research efforts. As a part of this work, we designed a new class of instruction set architectures (ISAs), called Explicit Data Graph Execution (EDGE), which renegotiate the boundary between hardware and software. EDGE ISAs contain a block-atomic execution model in which blocks consist of dataflow instructions. This model preserves sequential memory semantics, thus enabling a system that can expose greater instruction level concurrency without requiring explicit software parallelization. We constructed a custom 170 million transistor ASIC, a hybrid-dataflow instruction set (TRIPS ISA), TRIPS system circuit boards, a runtime system, performance evaluation tools, and a robust compiler that optimizes and translates C and Fortran programs to the TRIPS ISA. The TRIPS

microarchitecture is physically distributed into tiles connected in a nearest-neighbor fashion via microarchitecture networks (micronetworks). The distributed processing cores issue up to 16 instructions per cycle from an instruction window of up to 1024 instructions. The combination of the EDGE ISA and the TRIPS distributed microarchitecture is designed to exploit concurrency and reduce the influence of long wire delays by exposing the spatial nature of the microarchitecture to the compiler for optimization.

The details of the TRIPS architecture, microarchitecture, and compiler appear in prior publications. This paper describes a detailed performance analysis that explores how well the TRIPS compiler and hardware meets its goals of exploiting concurrency, hiding latency, and distributing control. Using the TRIPS hardware and detailed microarchitectural simulators to gather detailed statistics not available from the hardware, we compare the EDGE ISA, microarchitecture, and performance to modern processors using hand optimized and compiled benchmarks. We find that the EDGE ISA incurs a substantial overhead in total number of instructions fetched and executed, relative to conventional RISC architectures like the PowerPC, because of extensive predication and instruction overheads required by the dataflow model.

Our microarchitecture analysis shows that TRIPS can keep much of the instruction window full; compiled code shows an average of 450 total instructions in flight (887 peak for best benchmark) and hand optimized code shows an average of 630 (1013 peak). While much higher than conventional processors, the number of instructions in flight is less than the maximum of 1024 because the compiler does not completely fill blocks and the hardware experiences pipeline stalls and flushes due to I-cache misses, branch mispredictions, and load dependence mispredictions. A strength of the EDGE ISA and distributed control is that TRIPS requires less than half as many register and memory accesses as the PowerPC because it converts these into direct producer/consumer communications. Furthermore, the communicating instructions are usually on the same tile or an adjacent tile, which makes them power efficient and minimizes latency.

We compare the performance of TRIPS to the Intel Core 2, Pentium III, and Pentium 4 using hardware performance counters on compiled and hand-optimized programs. On the EEMBC suite, the Core 2 outperforms TRIPS compiled code by 30%. On SPEC-2000, TRIPS compiled code achieves only half the performance of the Core 2 on integer benchmarks but matches the performance of the Core 2 on floating point benchmarks. We find that TRIPS outperforms the Core 2 by an average factor of 3 measured in cycles on hand-optimized benchmarks and that the Core 2 outperforms the Pentium 3 and Pentium 4 by an additional factor of 2.

These experiments suggest that TRIPS-like processors have the capability to achieve substantial performance improvements over conventional microprocessors by exploiting concurrency. However, realizing this performance potential relies on the compiler to better expose concurrency and create large blocks of TRIPS instructions, as well as microarchitectural innovations in control distribution and branch prediction.

2 The TRIPS Processor Architecture

The foundations of the TRIPS ISA and microarchitecture were published in 2001 [14]. Between 2001 and 2004, we refined the architecture to a point where it could be realized in a silicon prototype and began to implement the TRIPS compiler. The TRIPS chip taped out in August 2006 and was shown to be fully functional (no known bugs) in the lab in February 2007. The TRIPS prototype is implemented in a 130nm ASIC technology and contains 170 million transistors. The simplest TRIPS system consists of four TRIPS chips, each with 2GB of local DRAM, connected to a motherboard. While the system is designed to be scalable to eight motherboards (64 processors), this paper examines a single TRIPS processor.

EDGE ISA: TRIPS implements the Explicit Data Graph Execution (EDGE) ISA [1], which was conceived with the goal of high-performance, single-threaded, concurrent, and distributed execution in which the microarchitecture maps compiler-generated dataflow graphs to a parallel execution substrate. Two defining features of an EDGE ISA are block-atomic execution [12] and direct instruction communication within a block, together enabling efficient hybrid dataflow execution. The TRIPS ISA aggregates up to 128 instructions into a single block that obeys a *block-atomic* execution model where each block is logically fetched, executed, and committed as a single entity. This amortizes the per-instruction bookkeeping over a large number of instructions and reduces branch predictions and register accesses. Furthermore, this model reduces the frequency of control decisions, providing the additional latency tolerance to make distributed execution practical. Blocks communicate through the register file and memory. Within a block, *direct instruction communication* delivers results from producer instructions to consumer instructions in dataflow fashion. Direct communication supports distributed execution within a block by eliminating accesses to a shared register file.

Figure 1 shows an example of a sequence of RISC code and the corresponding TRIPS EDGE code. The read and write instructions at the beginning and end of the TRIPS code encode the values that are injected from the register file into the dataflow instruction block (R0, R1) as well as the value that is ejected from the block (W0). Instruction operands within the block, such as `$t2`, are passed directly from producer to consumer without an intervening register file. Because the instructions encode their targets, rather than a register in a common register file, a 32-bit instruction encoding has room for at most two targets. When more targets are required, such as the value read in instruction R0, the program needs a `move` instruction (I0) to replicate the value flowing in the dataflow graph. The TRIPS code also shows that branch and non-branch instructions can be predicated. To enable the hardware to detect block completion, the execution model requires that all block outputs (register writes and stores) be produced regardless of the predicated path within the block. The `null` instruction produces a token that when passed through the `st` (store) indicates that the store output has been produced, but does not modify memory. In our experiments, we do not classify these dataflow execution

RISC Code

```
Label1:
ld   %3, 4(%2)
blez %3, Label2
ld   %5, 8(%2)
addi %3, %3, %5
st   %3, 4(%2)
Label2:
addi %4, %4, #-1
bgez %4, Label1
Label3:
```

TRIPS EDGE Code

```
.bbegin block1
R0: read  $t2, $g2
R1: read  $t4, $g4
I0: mov    $t3, $t2
I1: ld     $t5, 4($t2)
I2: ld     $t6, 8($t3)
I3: tlez   $t7, $t5
I4: addi_f<$t7> $t8, $t5, $t6
I5: null_t<$t7> $t8
I6: st      $t8, 4($t3)
I7: subi    $t9, $t4, #1
I8: teqz    $t10, $t9
I9: b_t<$t10> block3
I10:b_f<$t10> block1
W0: write $g4, $t9
.bend block1
.bbegin block3 ...
```

Dataflow Graph

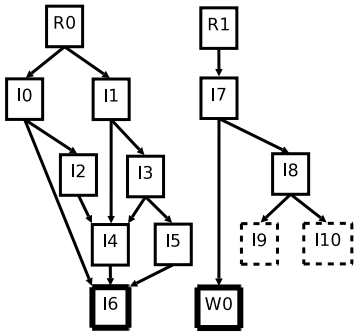


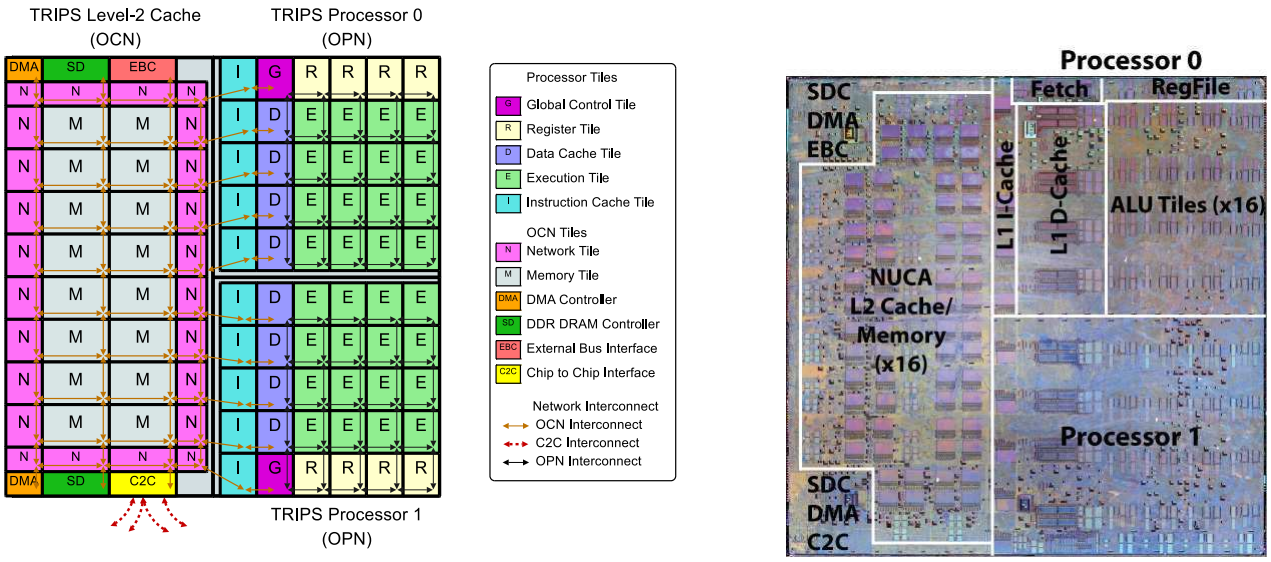
Figure 1: RISC code with its corresponding TRIPS EDGE code and dataflow graph

helper instructions as useful instructions when comparing to conventional ISAs. The dataflow graph at the right shows graphically the effective encoding of the instructions in the EDGE binary.

TRIPS Processor Microarchitecture: Because the goals of the TRIPS microarchitecture include scalability and distributed execution, it has no global wires, reuses a small set of components on routed networks, and can be extended to a wider-issue implementation without source recompilation or ISA changes. Figure 2 shows the tile-level block diagram and a die photo. Each TRIPS chip contains two processors and a secondary memory system, each inter-connected by one or more micronetworks. This paper analyzes the performance of a single TRIPS processor and its distributed microarchitecture on single-threaded codes. Here we summarize the basics of the architecture and a more detailed discussion of the microarchitecture can be found in [18].

Each of the processor cores uses five types of tiles: one global control tile (GT), 16 execution tiles (ET), four register tiles (RT), four data tiles (DT), and five instruction tiles (IT). The tiles communicate using six micronetworks that implement distributed control and data protocols. The main micronetwork is the operand network (OPN), which replaces a bypass network in a conventional superscalar processor. The OPN is a two-dimensional, wormhole-routed, 5x5 mesh network that delivers one 64-bit operand per link per cycle [6]. The other networks implement distributed instruction fetch, dispatch, I-cache refill, and completion/commit.

TRIPS fetches and executes each TRIPS block, *en masse*. The GT sends a block address to the ITs which deliver the computation instructions of the block to the reservation stations in the 16 execution tiles (ETs), 8 per tile as specified by the compiler; the ITs also deliver the register read/write instructions to reservation stations in the RTs. The RTs read values from the global register file and send them to the ETs, starting the dataflow execution. The GT instigates the commit protocol once each of the DTs and RTs receive all of the block outputs; the commit protocol updates the data caches and register file with the speculative state of the block. The GT uses its next block predictor (branch predictor) to begin executing the next block while previ-



(a) Block diagram (b) Die photo

Figure 2: TRIPS prototype

ous blocks are still executing. The TRIPS prototype can simultaneously execute up to eight 128-instruction blocks (one non-speculative, seven speculative) for an aggregate instruction window size of 1024 instructions.

At 130 nm, each TRIPS processor occupies approximately 92 mm^2 of a total chip area of 330 mm^2. If scaled down to 65 nm, a TRIPS core would be approximately 23 mm^2, similar to the 29 mm^2 of a Core 2 processor. A direct comparison is difficult because TRIPS is implemented with ASIC technology and lacks all of the hardware required to support an operating system. Nonetheless, TRIPS has a greater density of arithmetic units in a similar area and an architecture that enables greater issue width and instruction window scaling.

TRIPS Compiler: TRIPS has a fully functional compiler that can compile all of the C and Fortran SPEC2000 benchmarks [2, 11]. Using a machine-independent intermediate representation (IR), the compiler performs conventional optimizations such as inlining, unrolling, common subexpression elimination, scalar replacement, and some TRIPS-specific optimizations such as tree-height reduction to expose parallelism. With the renegotiation of the boundary between software and hardware, the compiler must perform two additional tasks compared with conventional compilers: block formation and instruction placement. The compiler aggregates basic blocks into larger, optimized TRIPS blocks using predication, tail duplication, and loop optimizations [11]. This process is similar to hyperblock formation, but TRIPS blocks have additional constraints that simplify the hardware [21]. The compiler exploits dataflow predication in the ISA to fuse code from multiple control paths into the same TRIPS block [22]. The compiler also determines which tile of the grid each instruction will dynamically execute when it is fetched and its operands are ready. This placement algorithm seeks to expose concurrency and minimize communication overheads (distance and contention) be-

System	Processor Speed	Memory Speed	Proc/Mem Speed Ratio	L1 Cache Capacity (D/I)	L2 Cache Capacity	Memory Capacity
TRIPS	366 MHz	200 MHz	1.83	32 KB / 80 KB	1 MB	2 GB
Core 2	1600 MHz	800 MHz	2.00	32 KB / 32 KB	2 MB	2 GB
Pentium 4	3600 MHz	533 MHz	6.75	16 KB / 12 K μ ops	2 MB	2 GB
Pentium III	450 MHz	100 MHz	4.50	16 KB / 16 KB	512 KB	256 MB

Table 1: Reference platforms

tween dependent instructions [2]. Placement optimizes performance without restricting functional portability as an EDGE binary can be run on different hardware topologies (number of tiles) without recompilation.

3 Evaluation Methodology

We evaluate the EDGE ISA and the TRIPS microarchitecture and compare its performance with conventional architectures using the on-board performance counters in the TRIPS hardware and in commercial platforms. We also use TRIPS simulators and a PowerPC simulator to gain deeper insights in Sections 4 and 5. All performance measurements in Section 6 are from the actual hardware.

TRIPS Prototype System: A TRIPS chip consists of two processors that share a 1 MB L2 static NUCA [9] cache and 2 GB of DDR Memory, but we use only one processor in all experiments. Each processor has a private 32 KB L1 data cache and a private 80 KB L1 instruction cache. The processor and memory speeds can be adjusted using a phased-lock loop (PLL); all of the experiments run the processor core at 366 MHz and the DRAM with 100/200 MHz DDR clocks. TRIPS system calls interrupt program execution, halt the processor, and are proxied to an off-chip commercial processor running Linux. Because the TRIPS cycle counters increment only when the processor is not halted, the program performance measurements ignore the time to process system calls. The tools we use to measure cycles in the commercial systems also exclude operating system execution time, thus providing a fair comparison.

Simulators: We use a functional TRIPS simulator and a low-level microarchitecture simulator to gather statistics not available from the hardware [24]. A Power PC functional simulator [17] produces statistics that measure loads, stores, and register accesses from *gcc* compiled PowerPC–AIX binaries.

Reference Platforms: We compare performance of the TRIPS prototype to three reference platforms from the Intel x86 product family (Pentium III, Pentium 4, and Core 2) using the PAPI software package to access the Intel processors' hardware counters [15]. Because each machine is implemented in a different process technology, we use cycle counts as the relative performance measure. Table 1 shows the platform configurations including processor and DDR DRAM clock speed and the memory hierarchy capacities. Cycle count is an imperfect metric because some architectures, particularly the Pentium 4, emphasize clock rate over cycle count. However, we expect that the TRIPS microarchitecture, with its partitioned design and no global wires, could be implemented in a clock rate equivalent to the Core 2, given a custom design and the same process

Suite	# of Benchmarks	Characteristics
Kernels	4	transpose (ct), convolution (conv), vector-add (vadd), matrix multiply (matrix)
VersaBench	3 of 10	bit and stream (fmradio, 802.11a, 8b10b)
EEMBC	30 of 30	Embedded benchmarks
Simple	15	Hand optimized versions of Kernels, VersaBench, and 8 EEMBC benchmarks
SPEC 2000 Int	10 of 12	All but gap and C++ benchmarks
SPEC 2000 FP	8 of 14	All but ammp, sixtrack, and 4 Fortran 90 benchmarks

Table 2: Benchmark suites

technology. Another pitfall with this comparison is that the relatively slow clock rate of TRIPS may make memory accesses less expensive relative to high clock-rate processors. To account for this, we under-clocked the Core 2 from 1.8 GHz to 1.6 GHz to equalize the processor/memory speed ratio to TRIPS. However, a slower clock has little effect on the measured applications because they are largely L2 cache resident.

Benchmarks: Table 2 shows the benchmark suites used, from simple kernels to complex uniprocessor workloads. We compiled these applications with the TRIPS C and Fortran compiler [21], and hand optimized the compiler-generated IR to study the performance potential of TRIPS. We extensively hand optimized four scientific kernels on TRIPS: matrix transpose (*ct*), convolution (*conv*), vector add (*vadd*), and matrix multiply (*matrix*). In addition, we hand placed *matrix* and *vadd* to achieve high performance. We hand optimized 3 stream and bit operation benchmarks from the VersaBench suite [16] and 8 medium-sized benchmarks from 30 EEMBC benchmarks [4]. The most complex benchmarks come from SPEC2000 and include 10 integer and 8 floating-point benchmarks [23]. Three SPEC programs that temporarily fail to build correctly with our toolchain are omitted, but will be included in a final version of this paper. SimPoint regions are used to select appropriate simulation points for a detailed evaluation of the SPEC benchmarks [20].

4 ISA Evaluation

This section examines how well the compiled and hand-optimized programs map into the ISA and characterizes block size, instruction overheads, and code size. We compare compiled and hand-optimized TRIPS programs to programs compiled for a RISC ISA (PowerPC) processor to quantify the relative overheads of the EDGE ISA. We also present the means of the EEMBC, SPEC INT, and SPEC FP suites to show the impact of the application characteristics. All of the data in this section was gathered through simulation because of the nature of the statistics needed.

4.1 TRIPS Block Size and Composition

A key parameter in specifying a block-atomic EDGE ISA is the block size. Early experience demonstrated that creating programs with average block sizes of 20+ instructions was not difficult with standard compiler transformations, and that larger blocks would lead to a larger instruction window, better amortize block overheads, and have the potential for better performance. We chose a maximum block size of 128 instructions as

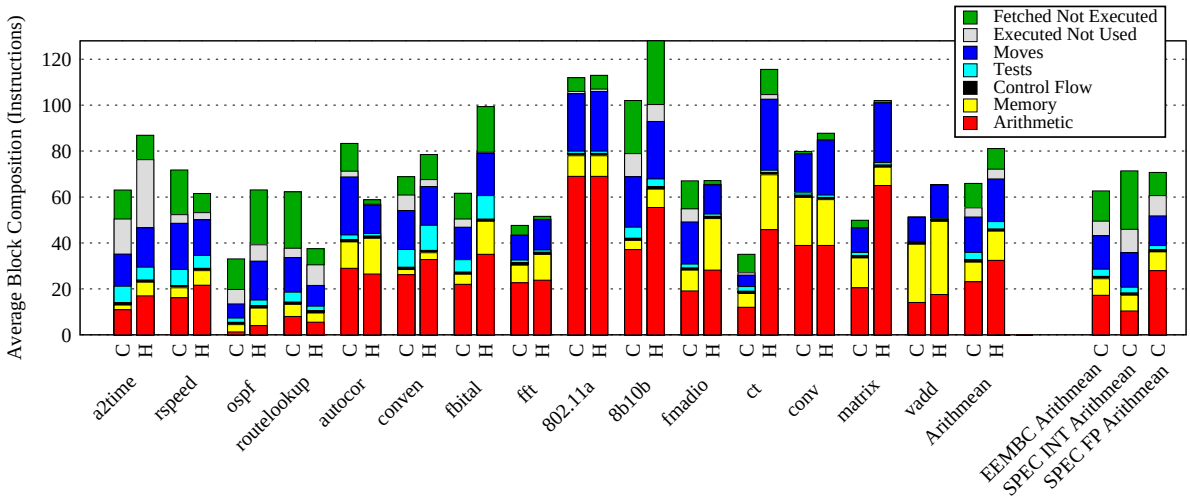


Figure 3: TRIPS block size and composition for compiled (C) and hand-optimized (H) benchmarks

an aggressive compiler target, to stress the software system to form larger blocks.

Figure 3 weighs each block's size by execution frequency to show the average block size, and breaks out the number of arithmetic instructions, memory instructions, branch/jump/call/return instructions, test instructions (used for branches and predication), and `move` instructions (used to fan out intermediate operands). The figure does not include the register read/write instructions since they reside in the block header and are not part of the 128 instructions. *Fetched Not Executed* instructions were fetched speculatively but never executed, either because they did not receive a matching predicate, or because they did not receive all of their operands due to predicated instructions earlier in the block's dataflow graph. *Executed Not Used* instructions were fetched and executed speculatively but whose values were not used due to predication later in the dependence graph.

For some programs, such as *a2time*, the number of mispredicated instructions accounts for half the total instructions within a block. *A2time* contains several nested `if/then/else` statements; to minimize the number of blocks executed, the compiler produces code that speculatively executes both the `then` and `else` clauses simultaneously within one block and inserts a predicate computation to select the correct outputs. Nonetheless, this aggressive predication can improve system performance because it eliminate branch mispredictions and enables the implementation of a pipeline with higher front-end fetch bandwidth.

The remainder of the instruction types, tests, control flow, memory, and arithmetic, are required for correct execution. The number of useful instructions (excluding `move` and mispredicated instructions) varies. Some programs with complex control have only 10 instructions per block while others with more regular control have as many as 80 instructions per block. To implement dataflow execution in a block, the EDGE ISA requires `move` instructions. First, since a TRIPS instruction has a fixed width, it can target at most two consumers. The compiler must therefore insert `move` instructions to fanout values consumed by more than two instructions. Second, predicate merge points, corresponding to *phi* merge nodes in the dataflow graph,

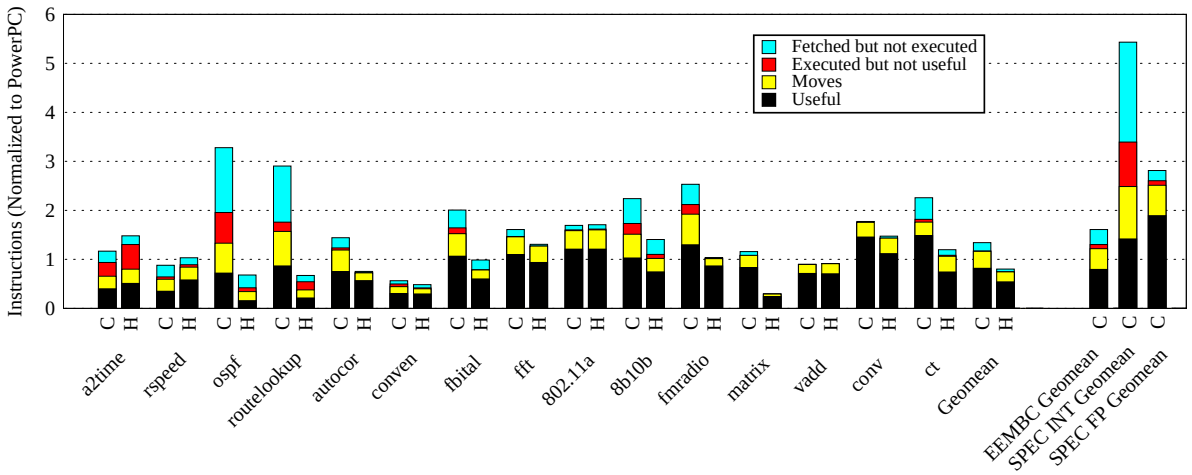


Figure 4: TRIPS instructions normalized to PowerPC for compiled (C) and hand-optimized (H) benchmarks

sometimes require predicated `move` instructions. The result is that `move` instructions account for nearly 20% of all instructions in a block, more than anticipated at the start of the design.

Compiled code has an average block size of 64 instructions, but with high variance, ranging from 30 to over 110 instructions. Hand optimizations to improve performance further increase block size. For example, the hand-optimized versions of *ospf* and *ct* have blocks two and three times larger than their respective compiled versions. These increases are often driven by instruction optimizations that decrease block size followed by opportunities to merge adjacent smaller blocks or by increasing unrolling factors to fill blocks. In summary, both the hand-optimized and compiled code utilize the aggressive 128-instruction block size to achieve average block sizes ranging from 20 to 128. To expose concurrency and aggressively speculate, the ISA overheads include `move` instructions and useless instructions, which are a significant fraction of the in-flight instructions.

4.2 TRIPS ISA versus PowerPC

To quantify the differences between an EDGE ISA and a popular RISC ISA, we compare to the PowerPC. Figure 4 shows fetched instruction counts on TRIPS normalized to PowerPC. Because of limitations in the current PowerPC infrastructure, the following graphs includes the mean of only 8 of the 18 SPEC benchmarks (5 INT and 3 FP). To compare expressive power of the compute instructions, the TRIPS instructions do not include register read/write instructions that appear in the block header, nor `NOP`s in underfull blocks. For both TRIPS and PowerPC, the instruction count omits incorrectly fetched instructions due to branch mis-predictions.

Not surprisingly, the number of useful instructions executed on TRIPS and PowerPC are similar because the TRIPS ISA is composed of RISC-style instructions. On compiled code, TRIPS tends to execute more instructions due to prototype simplifications, which introduce inefficiencies in constant generation and sign extension unrelated to its execution model. For hand-optimized benchmarks, TRIPS executes fewer instruc-

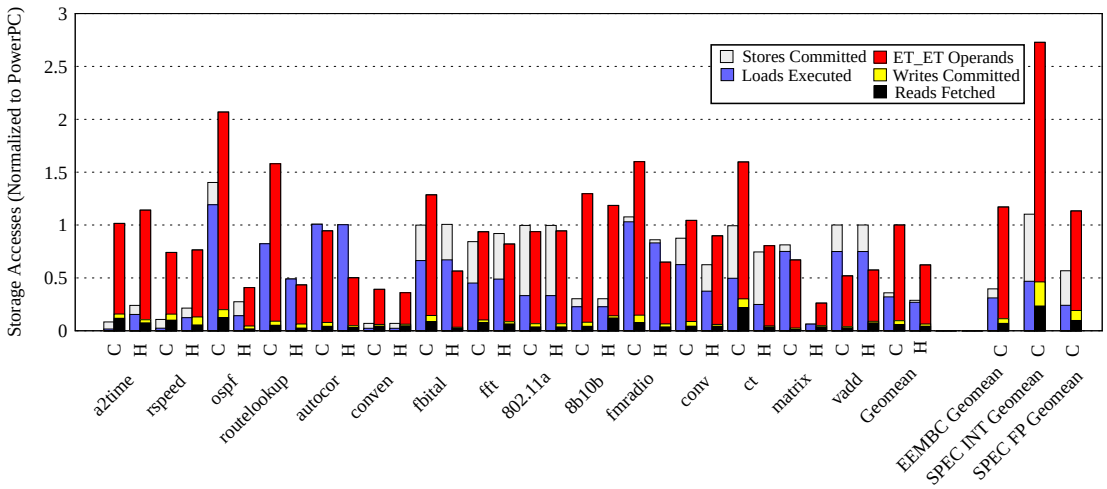


Figure 5: Storage accesses normalized to PowerPC for compiled (C) and hand-optimized (H) benchmarks

tions because its larger register set (128 registers) eliminates store/load pairs and because more aggressive unrolling exposes more opportunities for instruction reduction. The number of fetched but mis-predicated instructions varies across the benchmarks, depending on the degree of predication. Overall, TRIPS may need to fetch as many as 2–6 times more instructions than the PowerPC, due to aggressive predication.

4.3 Register and Memory Access

TRIPS inter-block communication is through registers and memory while intra-block communication is direct between instructions, reducing the number of accesses to registers and memory. The TRIPS prototype has a total of 128 registers spanning four register banks (32 registers per bank); each bank has only one read and one write port. The larger register file benefits the memory system as fewer register fills and spills are required. Eliminating store/load pairs ultimately improves performance as communication through registers is faster than communication through memory [13]. Compared to a conventional architecture, TRIPS replaces memory instructions for less expensive register reads and writes, and replaces register reads and writes for less expensive direct communication between producing and consuming instructions.

The left bar stack of each pair in Figure 5 shows the number of loads and stores on TRIPS normalized to the total number of loads and stores on the PowerPC. On average, TRIPS executes about half as many memory instructions as the PowerPC and as few as 15%, due to the bigger register file and direct communication. Several of the hand-optimized benchmarks have significantly fewer memory accesses than the compiled versions because they register allocate fields in structures and small arrays, whereas the compiler currently does not register allocate these. The right bar stack shows the number of register file reads, writes, and operand network communications on TRIPS normalized to the total number of register file reads and writes on the PowerPC. Because of direct operand communication, TRIPS requires only 10–20% of the register accesses

needed on the PowerPC. The top bar of the stack shows that the relative number of operands transmitted through direct communication far exceeds the number of register reads and writes on TRIPS.

Comparing each hand-optimized benchmark to its compiled counterpart on average, there are fewer register accesses, OPN communications, and memory accesses. The hand-optimized version aggressively register allocates more memory accesses by using programmer knowledge about pointer aliasing. It also has fewer instructions because it removes instructions with aggressive peephole optimizations and eliminates unnecessary sign extensions. On average, the sum of register reads, writes, and direct communication is about the same as the number of PowerPC register reads and writes. On some benchmarks, specifically SPEC INT, the temporary communication is large because of the distribution of predicates and communication of useless values by mis-predicated instructions. Figure 4 shows that the SPEC INT benchmarks fetch approximately half useless instructions, which leads to much more communication than on the PowerPC. In a conventional architecture, the register file broadcasts an instruction's result to other instructions. In TRIPS, fanout may require a tree of `move` instructions, which increase the number of copies of the original operand that are communicated.

4.4 Code Size

The TRIPS ISA increases dynamic code size over a PowerPC significantly. Each block has 128 32-bit instructions, a 128-bit header, 32 22-bit read instructions, and 32 six-bit write instructions. The compiler inserts `NOP`s when a block has fewer than 32 reads or writes or fewer than 128 instructions. `NOP`s consume space in the level-one I-cache but are not executed. We compared the dynamic code of TRIPS to the PowerPC by computing the number of unique instructions that are fetched during execution. The dynamic code size of TRIPS averages about 6 times larger than the PowerPC, but with a wide variance. The number of unique useful instructions for TRIPS is about 2 and 3 times that of the PowerPC, indicating that instruction replication due to TRIPS block optimizations accounts for about half of the code bloat. The `move` instructions and the block header (including useful register read and write instructions) account for about 30% of the total instructions.

The TRIPS prototype compresses underfull instruction blocks in memory and in the L2 cache down to 32, 64, or 96 instructions, depending on block capacity, which reduces the expansion factor over PowerPC to a factor of 4. Block compression in the instruction cache may unduly slow down instruction fetch or require more complex instruction routing from the instruction cache banks to the execution tiles. Experiments generally show a low instruction cache miss rate on small and medium sized benchmarks, but some SPEC benchmarks have miss rates in the range of 15–25%, indicating that instruction cache pressure is a serious problem for real applications. Fortunately, partitioned architectures, such as TRIPS, that bank the instruction cache can be easily designed with larger overall instruction caches to mitigate this type of cache pressure.

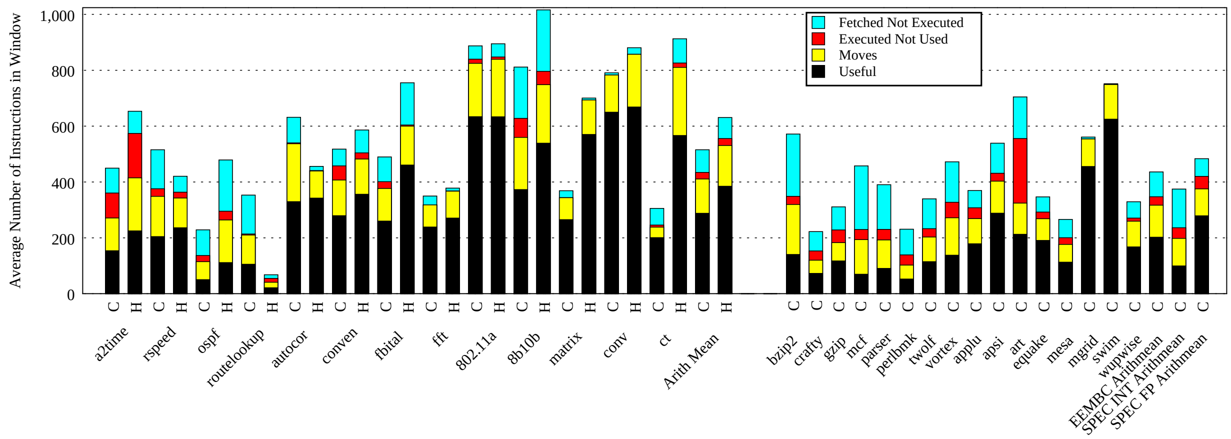


Figure 6: Average number of in-flight Instructions for compiled (C) and hand-optimized (H) benchmarks

5 Microarchitecture Evaluation

The primary goal of the TRIPS microarchitecture is to support a large instruction window with a partitioned design. One important aspect is the fraction of the instruction that is full, which depends on the TRIPS block predictor. Another aspect is the bandwidth of the partitioned memory system and the usage of the operand network. This section explores these unique aspects using detailed statistics gathered from simulation.

5.1 Filling a 1K Instruction Window

Each TRIPS block contains up to 128 instructions and the hardware can execute up to eight blocks concurrently so the maximum dynamic instruction window size, with full blocks and accurate speculation size, is 1024 instructions. Figure 6 shows the average number of TRIPS instructions in the window across a variety of hand and compiled benchmarks. This metric multiplies the average number of blocks in-flight (speculative and non-speculative) and the average number of instructions per block. Compiled codes, produce an average of 450 total instructions of which 200 are useful. The hand-optimized programs with larger blocks achieve a mean of 630 total instructions, more than 380 of which are useful. Compared with issue windows of 64 and 80 on modern superscalar processors, TRIPS exposes more concurrency, but at the cost of more communication.

In addition to predication, the principal speculation mechanisms in TRIPS are the store-load dependence predictor and the next-block predictor. When the load/store queue detects that a speculatively issued load has executed incorrectly, it flushes the block pipeline and enters the load into the dependence predictor's simple partitioned load-wait table in the data tile. For the SPEC benchmarks, the predictor is effective in part because the compiler reduces the number of loads and stores (as discussed in Section 4.3), resulting in fewer than one block flush per 2000 useful instructions, without overly constraining speculative load issue.

The TRIPS next-block predictor selects the next block to be fetched [18]. It consists of a 5 KB local/global

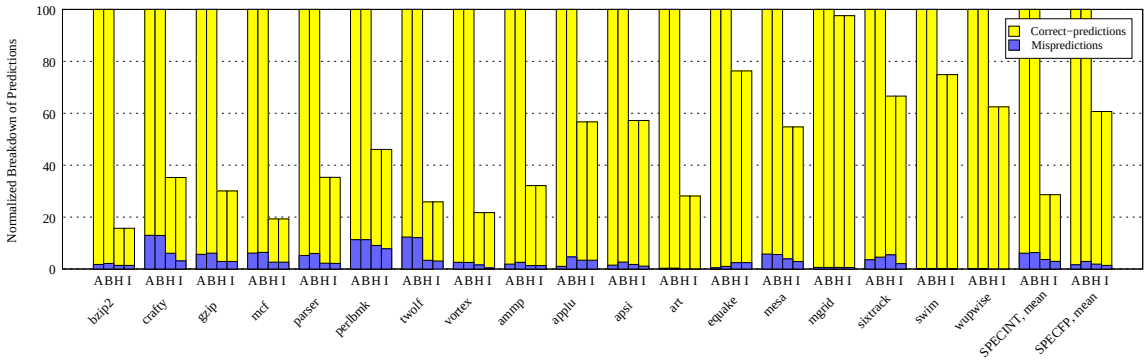


Figure 7: Prediction breakdown for Alpha 21264-like branch predictor on basic blocks (A), TRIPS block predictor on basic blocks (B), TRIPS block predictor on hyperblocks (H) and improved TRIPS block predictor on hyperblocks (I) normalized to total predictions made for basic blocks

tournament exit predictor that predicts which exit branch will be taken from the TRIPS block (one of up to eight) and a 5 KB multi-component target predictor that predicts the target address of this exit branch. Figure 7 shows the correct/misprediction breakdown for four different configurations: the first bar (*A*) shows the breakdown for an Alpha 21264-like conventional tournament branch predictor predicting TRIPS-compiled basic block code, the second bar (*B*) shows the TRIPS block predictor predicting basic block code, the third bar (*H*) shows the TRIPS prototype block predictor predicting hyperblock code, and the final bar (*I*) shows a "lessons learned" TRIPS block predictor that could be used in future designs constructed by scaling up the target predictor component to 9 KB. Each bar is normalized to the total number of predictions made for basic block code. The average MPKI (Mispredictions Per 1000 Instructions, omitting move and mispredicated instructions) observed for these four configurations on SPEC INT are 14.9, 14.8, 8.5 and 6.9 respectively. SPEC FP applications have an MPKI of 0.9, 1.3, 1.1 and 0.8 respectively.

Predicting TRIPS blocks rather than basic blocks can improve accuracy, because hard-to-predict branches are converted to predicates, and can degrade accuracy, because predication can obscure correlated branches in the history. Although the prototype predictor (*H*) has a higher misprediction rate than a conventional predictor (*A*), it has a lower MPKI because it makes fewer predictions (70% fewer on SPEC INT and 40% fewer on SPEC FP). The improved TRIPS predictor *I*, reduces SPEC INT MPKI by 19% and SPEC FP MPKI by 27%. More sophisticated multi-component long-history predictors [8, 19] could be used to improve the TRIPS predictor, as well as improving the efficiency of the exit encoding. Additionally, increasing the size of the currently small branch target buffer, call target buffer, and history will improve accuracy. Lower prediction accuracy has a significant effect on the utilization of the instruction window and, as discussed in Sections 5.3 and 6, has a strong correlation with performance. More aggressive next-block predictors would dramatically improve prediction accuracy, but may still fall short of modern branch prediction accuracies.

5.2 Feeds and Speeds

In this section, we explore the performance of the banked memory system and operand network, two defining features of the distributed TRIPS implementation. For the memory system, we developed kernels that saturate the bandwidth of each of the banks, providing insight into the types of optimizations required for memory-bound programs. For the operand network, we measure traffic load to determine how well the compiler's placement algorithm minimizes the distance between communicating instructions.

Memory System: The TRIPS prototype employs an address-partitioned memory system that divides the L1 data cache into four 8-KB, single-ported data banks and the L2 cache into sixteen 64-KB, single-ported memory banks. The table in Figure 8 shows the achieved memory bandwidth on the hardware at a core speed of 366 MHz for a hand-optimized vector add (`vadd`) kernel. With careful instruction placement, *vadd* can attain nearly 100% of the core's peak of four memory operations per cycle (10.5 GB/sec), indicating effective use of the partitioned L1 data cache. By adjusting the vector size of *vadd*, we constructed a microbenchmark with an access pattern to maximize the consumption of the L2 cache and main memory bandwidth. This program nearly reached the theoretical peak of the L2 and a majority of the main memory bandwidth provided by the dual DDR memory controllers. While the benchmark achieves only 57.8% of the maximum interface bandwidth, the vast majority of the loss is due to the memory controller protocol and not to the TRIPS design itself. Similar techniques and principles were used to hand-optimize dense matrix kernels [3] and lessons learned from these case studies were used to improve the compiler's instruction placement algorithms.

Operand Network: The Operand Network (OPN) connects the TRIPS processor tiles and transmits operands between execution tiles (ETs), the register file (RTs), and the data cache (DTs) [6]. The TRIPS compiler's instruction placer takes as input the tile topology and the dependencies between the instructions in each block. It optimizes the instruction placement to exploit concurrency and minimize the distance between dependent instructions along the program's critical path. The graph in Figure 8 displays the breakdown of the hop count for OPN traffic. On average, ET–ET operand traffic dominates the OPN and about half of the operands are bypassed locally within an ET resulting in an average operand hop count of 0.9. While an ideal instruction placement would use local bypassing for all operand communication (0 hops), the inherent trade-off between locality and concurrency combined with limited instruction storage per tile demands that many communicating instructions reside on different tiles. The ET–DT and ET–RT traffic typically requires more hops (and thus longer level-1 cache latency) because the DTs and RTs lie along the edge of the ET array. For example, *vadd* streams data from its L1 caches, yielding high ET–DT traffic, while *matrix* primarily uses data in its register file, yielding greater ET–RT traffic. These two microbenchmarks illustrate how the OPN supports highly divergent traffic patterns. We show the results of one SPEC benchmark, *gcc*, and the mean of

	L1 D-Cache to Processor	L2 to L1	Memory to L2
Peak Ops/ cycle	4 - 8byte requests	3.2 - 16 byte requests	1 - 64 byte request
Peak BW (GBytes/sec)	10.9	17.5	5.6
Achieved BW (GBytes/sec)	10.5	17.2	3.2
% of Peak	96.5%	98.5%	57.8%

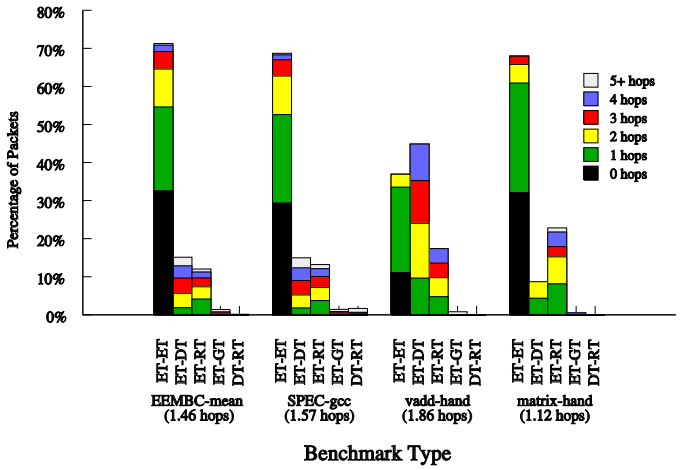


Figure 8: TRIPS bandwidths at 366MHz and operand network (OPN) profile with average hops per packet

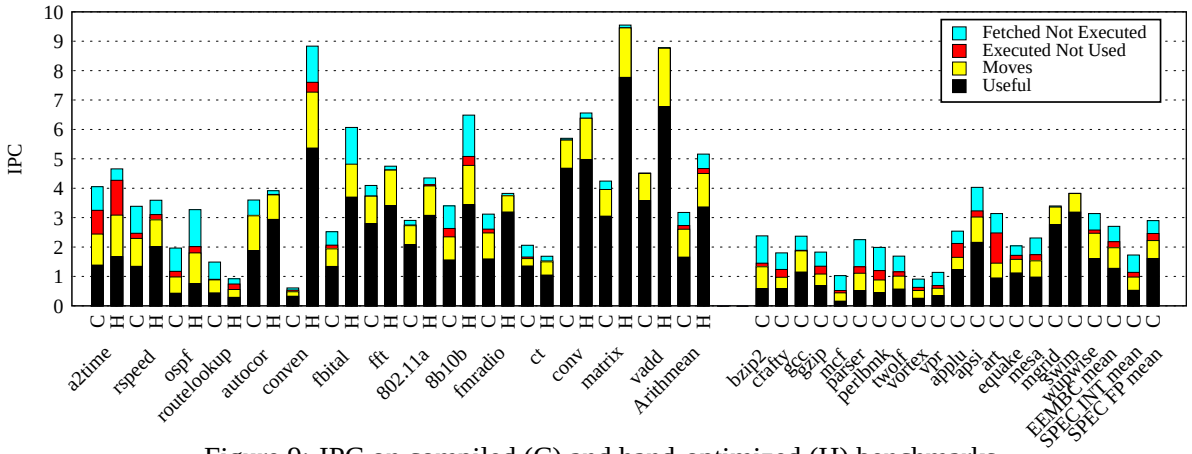


Figure 9: IPC on compiled (C) and hand-optimized (H) benchmarks

the EEMBC benchmarks to demonstrate that the load on the OPN is similar between these suites.

5.3 ILP Evaluation

The TRIPS prototype can execute up to 16 instructions per cycle, but can only sustain 16 IPC under ideal conditions: 8 blocks full of executed instructions, perfect next-block prediction, and no instruction stalls due to long-latency instructions. Actual IPC on the hardware is limited to 1/8 of the block size. Since the average block size of our hand-optimized benchmarks is 80 instructions, we could theoretically achieve at most an average IPC of 10 on them. Figure 9 shows the sustained IPC that TRIPS achieves across the benchmarks. While some applications are intrinsically serial (e.g., *routelookup*, which traverses a tree data structure serially), others reach 6 to 10 IPC, showing that the processor can take advantage of the greater ILP of these programs. The hand codes have an IPC 50% greater on average than their compiled counterparts, mostly due to better block optimization. The SPEC benchmarks have lower IPC, both because they have smaller average

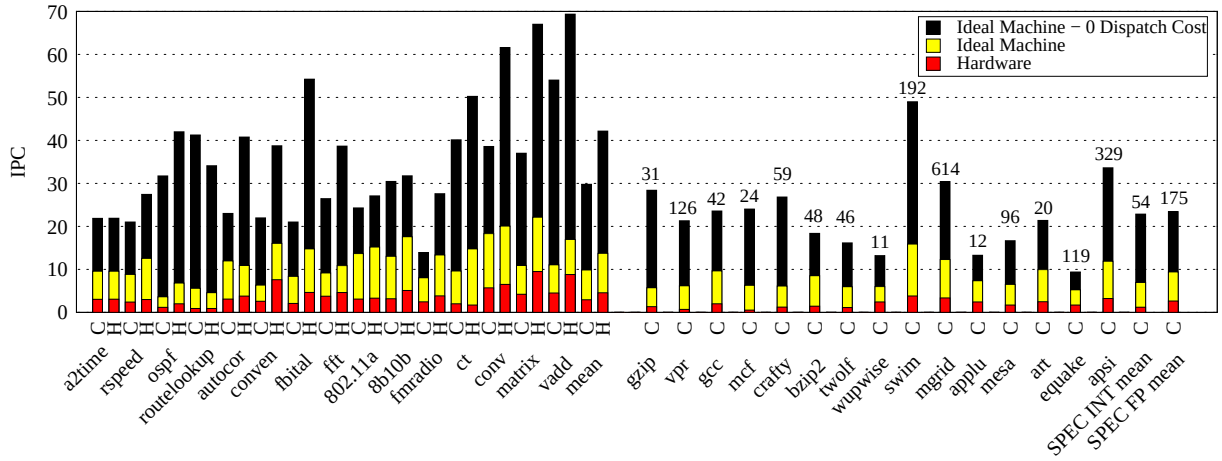


Figure 10: IPC for TRIPS and an ideal EDGE machine, numbers shown are IPC for SPEC benchmarks for an ideal machine with a 128K instruction window

block sizes, and more flushes due to branch mispredictions and i-cache misses.

To understand the theoretical ILP capability of EDGE architectures, we conducted a limit study using an idealized EDGE machine with perfect prediction, perfect predication, perfect caches, infinite execution resources, and a zero-cycle delay between tiles. It, like TRIPS, has a window size of 1K instructions and a dispatch and fetch cost that only allows a new block to be started once every eight cycles. Figure 10 shows that on average this ideal machine only outperforms the prototype by roughly a factor of 2.5, indicating only moderate room for improvement due to low inherent application ILP, the dispatch cost, and limited window size. Simulating this ideal machine with a zero-cycle dispatch cost increases the IPC on average by a factor of five. However, eliminating only the dispatch delay on TRIPS improves performance by only 10%, which indicates that dispatch is not the primary bottleneck on the hardware. We also annotate the top of the SPEC bars with the IPC for the ideal machine with a window of 128K instructions and a dispatch cost of zero cycles. The SPEC benchmarks have a wide range of available ILP, with most benchmarks around 50 IPC but some FP benchmarks having IPCs in the hundreds. The simple benchmarks have a similar range of IPCs with several such as *802.11a* and *8b10b* that are inherently serial and do not exceed 15; others such as *vadd* and *fmradio* are concurrent but are resource limited on the hardware resulting in IPCs of 1000 and 500 respectively on the ideal machine with a 128K window. This study reveals that while the hardware has room to improve, the amount of ILP currently available to TRIPS is limited and that larger window machines have the potential to further exploit ILP.

6 TRIPS Performance versus Commercial Platforms

This section compares TRIPS to conventional processors using hand-optimized benchmarks to show the potential of TRIPS and compiled benchmarks to show the current state of the compiler. TRIPS code is compared

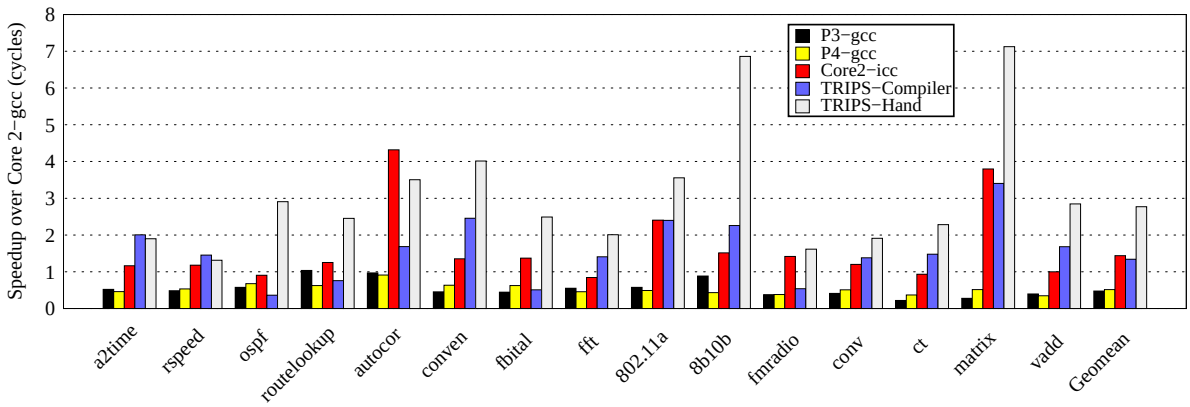


Figure 11: Speedup of simple benchmarks relative to Core 2-gcc.

to benchmarks compiled with both the GNU C compiler (gcc) as well as the native compiler (icc) on the reference machines. Data in this section is obtained from hardware performance counters.

Simple Benchmarks: Figure 11 shows cycle counts for TRIPS hand-optimized code, TRIPS compiled code, icc-compiled code for the Intel Core 2, and gcc-compiled code for the Intel Core 2, Pentium 4, and Pentium III, normalized to the Core 2 using gcc. The TRIPS compiler achieves an average 1.5x speedup over the Core 2, but does not perform as well on three of the benchmarks. Benchmarks with smaller speedups like *rspeed* are sequential algorithms that do not benefit from increased execution bandwidth or deep speculation. The benchmarks that show the largest speedups, such as *matrix* and *8b10b*, typically have substantial parallelism exposed by the large window on TRIPS. The TRIPS hand-assembled code always outperforms the Core 2, with an average 2.9x speedup.

The performance differences between TRIPS compiled code and TRIPS hand-assembled code are primarily due to more aggressive block formation, unrolling, and scalar replacement. For example, *8b10b* benefits from unrolling the innermost loop of the kernel to create a full 128-instruction block and from register allocating a small lookup table. In *fmradio*, the hand-optimized code fuses loops that operate on the same vector, and uses profile information to exclude infrequently taken paths through the kernel.

To completely remove the influence of the compiler and show the ability of TRIPS to exploit a large number of functional units, we compare a TRIPS hand-optimized matrix multiply [3] to the state-of-the-art hand-optimized assembly versions of GotoBLAS Streaming Matrix Multiply Libraries on Intel platforms [5]. The following are the best published results from library implementations for conventional platforms, which differ from the experimental numbers from compilation found in Figure 11. The performance across platforms, measured in terms of FLOPS Per Cycle (FPC), ranges from 1.87 FPC on the Pentium 4 to 3.58 FPC on the Core 2 using SSE. The TRIPS application is able to achieve 5.20 FPC without the benefit of SSE, which is 40% greater than the best optimized code on the Core 2.

SPEC CPU2000: Figure 12 compares the performance of the SPEC2000 benchmarks on TRIPS with

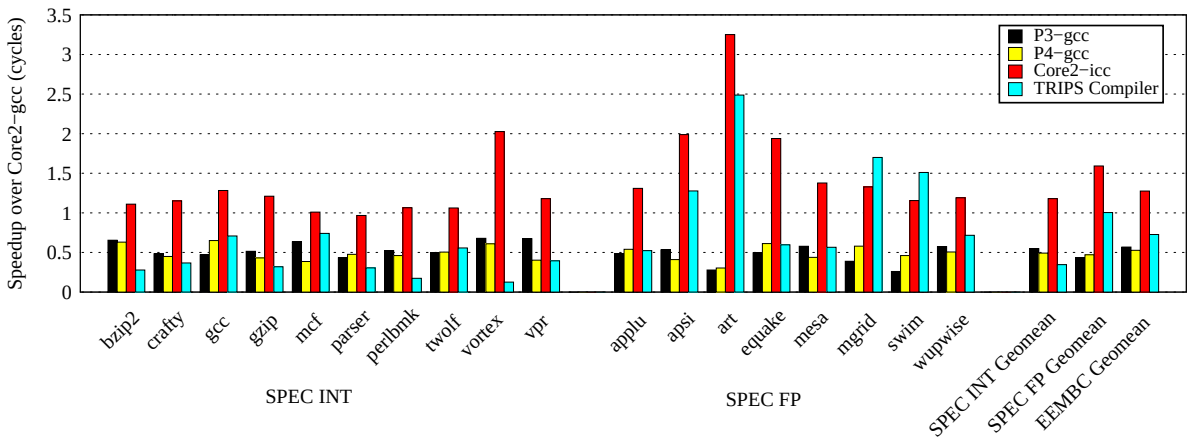


Figure 12: Speedup of SPEC benchmarks relative to Core 2-gcc.

the reference platforms. For consistency with our simulated results, we report performance over a SimPoint region, but we see similar results on the full applications. On the Intel platforms we use both icc and gcc, to identify the effect of platform-specific optimizations. The quality of scalar optimization in gcc is more similar to the TRIPS compiler than icc, since the TRIPS compiler is an academic research compiler that targets multiple architectures. Consequently, we normalized performance to the Core 2 using the gcc compiler.

TRIPS performance is much lower on the SPEC benchmarks than on the simple benchmarks shown in Figure 11. While floating point performance is on par with Core 2-gcc (Core 2-icc achieves a speedup of 1.6 over TRIPS), integer performance is less than half that of the Core 2. Table 3 shows several events that have a large effect on performance: conditional branch mispredictions, call-return mispredictions, I-cache misses, and load flushes for TRIPS, normalized to events per 1000 useful TRIPS instructions. Also shown are the branch mispredictions and I-cache misses for the Core 2, normalized to the same 1000 TRIPS instruction-baseline, which makes possible a cross-ISA comparison. The two rightmost columns capture the effective reduction in instruction window size due to the pipeline flushes discussed above. The second column from the right provides an estimate of the number of useful instructions that would be in the window if flushes and I-cache misses did not occur. The rightmost column shows the measured average useful TRIPS instructions in the window, first shown in Figure 6.

Several of the SPECINT benchmarks have frequent I-cache misses, such as *crafty*, *perlbmk*, *twolf*, and *vortex*. These benchmarks are known to stress the instruction cache, and the block-based ISA exacerbates the miss rate because of both the TRIPS code expansion and the compiler's inability to fill the fixed-size 128-instruction blocks. *Crafty*, *perlbmk*, and *vortex* also have an unusually high number of call/return mispredictions, due to an insufficiently tuned call and branch target buffer in TRIPS. All of these factors reduce the utilization of the instruction window; for example, *perlbmk* has only an average of 52 useful instructions in flight, out of a possible 155 based on the average block size. While the TRIPS call/return flushes and I-cache misses cause

	Per 1000 useful TRIPS instructions						Average useful block size * 8	Average useful insts in flight
	Core 2 cond. br. misses	TRIPS cond. br. misses	TRIPS call/ret misses	Core 2 I-cache misses	TRIPS I-cache misses	TRIPS load flushes		
bzip2	5.5	4.0	0	0	0	0.02	183.5	140.3
crafty	5.9	5.5	3.5	2.55	15.8	0.48	154.0	72.5
gcc	0.2	0.2	0.1	0	0.3	0.01	283.8	271.8
gzip	6.2	3.1	1.4	0	0	0.06	176.9	117.1
mcf	23.9	9.1	0.8	0	0	0.19	113.0	69.8
parser	6.0	2.6	1.8	0	1.1	0.12	118.7	90.3
perlbmk	2.5	1.2	11.1	0.01	3.2	0.22	122.4	52.6
twolf	13.9	4.9	2.1	0	9.2	0.57	182.7	114.6
vortex	0.4	0.6	3.2	0.48	8.1	0.44	155.3	137.6
vpr	13.8	4.3	0.9	0	0	0.06	176.8	—
applu	0.2	1.3	0	0	0	0.14	216.8	178.9
apsi	0	0.7	0	0.2	3.3	0.16	383.0	107.4
art	0.5	0	0	0	0	0.01	218.3	212.6
equake	0.3	0.6	0	0	0	0.04	218.8	190.7
mesa	2.2	2.8	0.2	0.01	7.9	0.08	192.8	112.9
mgrid	0.1	0.1	0	0	0	0.01	469.1	455.2
swim	0	0	0	0.01	0	0	644.2	624.7
wupwise	0	0	0	0.01	0	0	202.5	167.9

Table 3: TRIPS performance counter data for the SPEC benchmarks.

serious performance losses, branch mispredictions are competitive with the Core 2 and load dependence mispredictions are infrequent. The benchmarks that are most able to keep many useful instructions in the window compare best to Core 2, such as *art*, *mgrid*, and *swim*. These benchmarks are well known to be good targets for extracting parallelism, and show good performance with little compiler or microarchitectural tuning.

7 Lessons Learned

The prototyping effort's goals were twofold: to determine the viability of EDGE technology and to learn the right way to build an EDGE-based machine. While this effort was successful in answering some of the high-level questions about EDGE designs, it did provide significant insight about how to (and how not to) build an EDGE processor. This design and evaluation effort taught the following specific lessons about how this class of architectures should be built:

EDGE ISA: Prototyping has demonstrated that EDGE ISAs can support power-efficient, large-window, out-of-order execution with less complexity than an equivalent superscalar processor. However, the TRIPS ISA had several significant weaknesses. Most serious was the limited fanout of the `move` instructions, which results in far too many overhead instructions for high-fanout operations. The ISA needs support for limited broadcasts of high-fanout operands. In addition, the binary overhead of the TRIPS ISA is too large. The 128-byte block header, with the read and write instructions, adds too much per-block overhead. Future EDGE ISAs should shrink the block header to no more than 32 bytes, and must support variable-sized blocks in the L1 I-cache to reduce the NOP bloat, despite the resultant increase in microarchitectural complexity.

Compilation: The TRIPS compiler and prototype has shown that correct EDGE code can be generated, even for complex integer applications. The hand optimizations that proved effective are largely mechani-

cal, indicating that a production EDGE compiler could achieve much of that improvement. Because of the instruction-level block constraints, we determined that structural optimizations, such as loop unrolling and hyperblock formation, should occur in the back end after code generation. In general, the ISA model faces several difficult compilation challenges, the most significant of which is forming large blocks in control-intensive code. The major challenge is frequent function calls that cut blocks too early; inlining cannot solve this problem because it occurs well before block formation, typically in the front end. A second critical problem is allocating as many variables in registers as possible; the best hand-generated code replaced store-load pairs with a intra-block temporary communications, producing tighter code and higher performance. Effective interprocedural alias analysis is required to discover sufficient opportunities for this optimization.

Microarchitecture: The TRIPS prototype demonstrates that a microarchitecture with distributed protocols is feasible, and the fully functional first silicon indicates that tiled architectures benefit from increased design and validation productivity. A positive result was that, in general, the distributed block control protocols (fetch, dispatch, commit, flush) are not on the critical path. However, a number of artifacts in the microarchitecture resulted in significant performance losses. Most important was traffic on the operand network, which averaged just under one hop per operand. That amount of communication resulted in both significant OPN contention and communication cycles on the critical path. Spreading a block's instructions among all execution tiles caused too much intra-block communication. Follow-on microarchitectures must re-map instructions, in coordination with the compiler, so that most instruction-to-instruction communication occurs on the same tile. The second most important lesson was that performance losses due to the evaluation of predicate arcs was occasionally high, since arcs that could have been predicted as branches are deferred until execution. Future EDGE microarchitectures must support predicate prediction to evaluate the most predictable predicate arcs earlier in the pipeline. Third, the primary memory system must be distributed among all of the execution tiles; the cache and register bandwidth along one edge of the execution array was insufficient for many bandwidth-intensive codes. Finally, a minor design flaw in the prototype was making the call/return predictors too small, which should be enlarged in future microarchitectures. Improvements in branch and dependence predictors will also result in higher performance for all microarchitectures, including EDGE designs.

8 Conclusions

At its inception, the TRIPS design and prototyping effort tried to answer the following high-level questions: (1) whether an effective distributed, EDGE-based processor could be built using a tiled approach, (2) whether EDGE ISAs form a manageable compiler target, and (3) whether an EDGE-based processor can support improved general-purpose, single-threaded performance. This evaluation shows that the TRIPS ISA and microarchitecture are in fact feasible to build, resulting in a tiled design that exploits out-of-order execution over

a window of many hundreds of instructions. Despite the inter-tile routing latencies, the combination of the large window, dynamic issue, and highly concurrent memory system permits TRIPS to sustain up to 10 IPC, showing average of 3x speedup over a Core 2 processor across a diverse set of hand-optimized kernels.

For small, regular codes, the TRIPS compiler is able to generate codes that run in fewer cycles than state-of-the-art industrial designs, indicating that high-quality, compiler-generated EDGE code is feasible to produce. Even though the compiled code performs less well than the hand-generated code, the experience of converting compiled code into hand-optimized code indicates that most of that performance gap can be eliminated by mechanical transformations in an aggressive optimizing compiler.

However, the compiled cycle counts on major benchmarks, such as SPECINT and SPECFP, are not competitive with industrial designs, despite the greater computational resources present in TRIPS. On compiled SPEC2000 benchmarks, the TRIPS prototype achieves 60% of the performance of a Core 2 running SPEC2000 compiled at full optimization with gcc. While coming within 50% of an industry leader using a system built by fewer than twenty people is a significant technical achievement, it does not indicate that this model can substantively outperform the current industrial designs on large, complex applications. Even if this level of performance is increased moderately, the gains are likely too small to justify a switch to a new class of ISAs for high-end commercial systems. These limitations are due partially to inefficiencies in the ISA and microarchitecture, but may also result from a fundamental mismatch between certain program features and EDGE ISAs. For example, benchmarks with many indirect jumps, or unusually complex call graphs with many small functions, may be difficult to build into large blocks without a code size explosion.

The prototyping effort was intended to learn the lessons necessary to build the best possible EDGE-based designs. Support for variable-sized blocks, partial broadcast of operands, predicate prediction, a more distributed/scalable memory system, smaller block headers, and alternate mappings of instructions to tiles all emerged as important and necessary features of future EDGE-based designs. In addition, since not all codes have high concurrency, future EDGE-based microarchitectures must allow adaptive granularity, providing more efficient small configurations when larger configurations provide little performance benefit [10]. Currently, we project that these improvements will not enable large speedups of EDGE designs over high-end commodity systems, although they will result in large gains over the TRIPS prototype. We believe that these designs will show the most benefit over industrial designs in the five-to-ten watt space, and may be sufficiently faster in that space to justify adoption.

References

[1] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[2] K. Coons, X. Chen, S. Kushwaha, D. Burger, and K. McKinley. A Spatial Path Scheduling Algorithm for EDGE Architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–140, October 2006.

[3] J. Diamond, B. Robatmili, S. W. Keckler, K. Goto, D. Burger, and R. van de Geijn. High Performance Dense Linear Algebra on Spatially Partitioned Processors. In *Symposium on Principles and Practice of Parallel Programming*, pages 63–72, February 2008.

[4] `http://www.eembc.org`.

[5] K. Goto and R. A. van de Geijn. Anatomy of High-Performance Matrix Multiplication. *ACM Transactions on Mathematical Software*, 34(12):4–29, May 2008.

[6] P. Gratz, K. Sankaralingam, H. Hanson, P. Shivakumar, R. McDonald, S. W. Keckler, and D. Burger. Implementation and Evaluation of a Dynamically Routed Processor Operand Network. In *International Symposium on Networks-on-Chip*, pages 7–17, May 2007.

[7] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, July 2008.

[8] D. Jiménez. Piecewise Linear Branch Prediction. In *International Symposium on Computer Architecture*, pages 382–393, June 2005.

[9] C. Kim, D. Burger, and S. W. Keckler. An Adaptive Non-Uniform Cache Structure for Wire-Dominated On-Chip Caches. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, October 2002.

[10] C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, S. W. Keckler, and D. Burger. Composable Lightweight Processors. In *International Symposium on Microarchitecture*, pages 381–294, December 2007.

[11] B. Maher, A. Smith, D. Burger, and K. S. McKinley. Merging Head and Tail Duplication for Convergent Hyperblock Formation. In *International Symposium on Microarchitecture*, pages 65–76, December 2006.

[12] S. Melvin and Y. Patt. Enhancing Instruction Scheduling With a Block-Structured ISA. *International Journal on Parallel Processing*, 23(3):221–243, June 1995.

[13] A. Moshovos and G. S. Sohi. Speculative Memory Cloaking and Bypassing. *International Journal of Parallel Programming*, 27(6):427–456, December 1999.

[14] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *International Symposium on Microarchitecture*, pages 40–51, December 2001.

[15] PAPI: Performance Application Programming Interface. `http://icl.cs.utk.edu/papi`.

[16] R. M. Rabbah, I. Bratt, K. Asanovic, and A. Agarwal. Versatility and VersaBench: A New Metric and a Benchmark Suite for Flexible Architectures. Technical Report TM-646, Laboratory for Computer Science, Massachusetts Institute of Technology, June 2004.

[17] K. Sankaralingam, R. Nagarajan, S. Keckler, and D. Burger. SimpleScalar Simulation of the PowerPC Instruction Set Architecture. Technical Report TR-00-04, Department of Computer Sciences, The University of Texas at Austin, February 2001.

[18] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed Microarchitectural Protocols in the TRIPS Prototype Processor. In *International Symposium on Microarchitecture*, pages 480–491, December 2006.

[19] A. Seznec and P. Michaud. A Case for (Partially) TAgged GEometric History Length Branch Prediction. *Journal of Instruction-Level Parallelism*, Vol. 8, February 2006.

[20] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, September 2001.

[21] A. Smith, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, K. S. McKinley, and J. Burrill. Compiling for EDGE Architectures. In *International Symposium on Code Generation and Optimization*, pages 185–195, March 2006.

[22] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. Dataflow Predication. In *International Symposium on Microarchitecture*, pages 89–102, December 2006.

[23] `http://www.spec.org`.

[24] B. Yoder, J. Burrill, R. McDonald, K. Bush, K. Coons, M. Gebhart, M. Govindan, B. Maher, R. Nagarajan, B. Robatmili, K. Sankaralingam, S. Sharif, A. Smith, D. Burger, S. W. Keckler, and K. S. McKinley. Software Infrastructure and Tools for the TRIPS Prototype. In *Workshop on Modeling, Benchmarking and Simulation*, June 2007.